VISION TRANSFORMER FOR MULTIVARIATE TIME-SERIES CLASSIFICATION (VITMTSC)

Anonymous authors
Paper under double-blind review

ABSTRACT

Multivariate Time-Series Classification (MTSC) is an important issue in many disciplines because of the proliferation of disparate data sources and sensors (economics, retail, health, etc.). Nonetheless, it remains difficult due to the high-dimensionality and richness of data that is regularly updated. We present a Vision Transformer for Multivariate Time-Series Classification (VitMTSC) model that learns latent features from raw time-series data for classification tasks and is applicable to large-scale time-series data with millions of data samples of variable lengths. According to our knowledge, this is the first implementation of the Vision Transformer (ViT) for MTSC. We demonstrate that our approach works on datasets ranging from a few thousand to millions of samples and achieves close to the state-of-the-art (SOTA) results on open datasets. Using click-stream data from a major retail website, we demonstrate that our model can scale to millions of samples and vastly outperform previous neural net-based MTSC models in real-world applications. Our source code is publicly accessible at `https://github.com/mtsc-research/vitmtsc` to facilitate further research.

1 INTRODUCTION

Deep neural networks (DNNs) have shown significant effectiveness with both text (Lin et al., 2021) and images (Li et al., 2021). The availability of standard DNN architectures that effectively encode raw data into meaningful representations, leading in good performance on new datasets and associated tasks with little effort, is a crucial factor enabling advancement. In image understanding, for instance, variations of residual convolutional networks, such as ResNet (He et al., 2015), exhibit relatively outstanding performance on new image datasets or somewhat different visual recognition problems, such as segmentation. Despite being the most prevalent sort of data in the actual world, tabular data has only recently been the focus of deep learning research, such as TabNet (Arik & Pfister, 2019) and TabTransformer (Huang et al., 2020). Time-Series data is a special case of tabular data. An intrinsic characteristic of time-series is that observations within time-series are statistically dependent on assumed generative processes (Löning et al., 2019). For example, the likelihood of a user streaming a particular movie X, is dependent on whether they have streamed movie X before as well as latent factors like, the time duration since the last streaming of movie X, number of movies from the same genre/director/actor as X that the user has streamed, cost of streaming, promotions on movie X, etc. Due to this reliance, time-series data does not readily fit inside the traditional machine learning paradigm for tabular data, which implicitly implies that observations are independent and identically distributed (i.i.d.).

We consider time-series, input to the model, to be composed of (a) time-points at which they are observed, and (b) observations (or time-series datapoints or TS datapoints) at those time-points. We denote, a time-series object that contains N samples as $\mathbf{X} = [\mathbf{X}_{t_1}, \mathbf{X}_{t_2}, ..., \mathbf{X}_{t_N}]$, where elements of the sequence are observation(s) at time-points t_1, t_2, ..., t_N respectively. The time-series data may be (a) Univariate time-series (UTS), in which a single variable is observed over time, or (b) Multivariate time-series (MTS), in which two or more variables are recorded over time. We denote an individual multivariate time-series datapoint as $\mathbf{X}_{t_K} = [\mathbf{X}^1_{t_K}, \mathbf{X}^2_{t_K}, ..., \mathbf{X}^M_{t_K}]^T$ for 1, 2, ..., M distinct variables/features observed at each time-point. Please note in large commercial datasets, an individual $\mathbf{X}^n_{t_K}$ categorical variable can have millions of distinct values (refer to Table 2).

In this paper, we will focus on Multivariate Time-Series Classification (MTSC) (Ruiz et al., 2021), which is an area of machine learning interested in assigning labels to MTS data. In non-DNN based MTSC methods, the data is first converted to i.i.d. via feature transformation, and then traditional classifiers are applied (Ruiz et al., 2021). Only a few available approaches for MTSC consider DNNs for this task (Fawaz et al., 2018) (refer to Section 2). One advantage of using DNN for MTSC is predicted efficiency benefits, particularly for large datasets (Hestness et al., 2017). In addition, DNNs offer gradient descent (GD) based end-to-end learning for time-series data, which has numerous advantages: (a) quickly encoding multiple data kinds, including images, tabular, and time-series data; (b) eliminating the requirement for feature engineering, which is a fundamental part of non-DNN MTSC approaches; (c) learning from streaming data; and perhaps most crucially (d) end-to-end models enable representation learning, which enables many valuable applications including including data-efficient domain adaptation (Goodfellow et al., 2016), generative modeling (Radford et al., 2015), semi-supervised learning (Dai et al., 2017), and unsupervised learning (Karhunen et al., 2015).

We present Vision Transformer for Multivariate Time-Series Classification (**VitMTSC**), a ViT-based pure transformer approach to MTSC, to model long-range contextual relationships in MTS data. The VitMTSC model accepts raw time-series data as input and is trained with GD based optimization to facilitate flexible end-to-end learning. We evaluate VitMTSC using both publicly available and proprietary datasets. The majority of algorithms in the MTSC literature have been evaluated on open-source UEA datasets (Ruiz et al., 2021). We evaluate VitMTSC on five UEA datasets (Table 1) and demonstrate that VitMTSC achieves comparable performance to the current SOTA methods (Table 3). Using click-stream data (Table 2) from a popular e-commerce site, we show that VitMTSC can meet the needs of modern datasets and performs much better than current DNN methods (Table 4) on both small and large real-world datasets (refer to Section 4 for experiment details).

2 Related Work

Since AlexNet (Krizhevsky et al., 2012), deep convolutional neural networks (CNNs) have advanced the state-of-the-art across many standard datasets for vision problems. At the same time, the most prominent architecture of choice in sequence-to-sequence modeling is the Transformer (Vaswani et al., 2017), which does not use convolutions, but is based on multi-headed self-attention (MSA). The MSA operation is particularly effective at modelling long-range dependencies and allows the model to attend over all elements in the input sequence. This is in stark contrast to convolutions where the corresponding "receptive field" is limited, and grows linearly with the depth of the network. The success of attention-based models in Natural Language Processing (NLP) has inspired approaches in computer vision to integrate transformers into CNNs (Wang et al., 2017; Carion et al., 2020), as well as some attempts to replace convolutions completely (Parmar et al., 2018; Bello et al., 2019; Ramachandran et al., 2019). With the Vision Transformer (ViT) (Dosovitskiy et al., 2020), however, a pure-transformer-based architecture has recently outperformed its convolutional counterparts in image classification. Since then, the ViT model has also been applied to other domains, for example, video classification (Arnab et al., 2021) and many more variants have been proposed [1].

In the last two decades, time-series data has proliferated a plethora of fields including economics (Wan & Si, 2017), health and medicine (Gharehbaghi et al., 2015), scene classification (Nwe et al., 2017), activity recognition (Chen et al., 2020), traffic analysis (Yang et al., 2014), click-stream analysis (Jo et al., 2018) and more. DNN is used in modeling time-series tasks, such as forecasting (Benidis et al., 2020; Lim & Zohren, 2021; Torres et al., 2021), classification (Wang et al., 2016; Fawaz et al., 2018; Zou et al., 2019; Fawaz et al., 2019), anomaly detection (Blázquez-García et al., 2020; Choi et al., 2021), and data augmentation (Wen et al., 2021). Similarly, Transformers (Wen et al., 2022) have been used in Time-Series forecasting (Li et al., 2019; Zhou et al., 2020; 2022), anomaly detection (Xu et al., 2021; Tuli et al., 2022), classification (Rußwurm et al., 2019; Zerveas et al., 2021; Yang et al., 2021).

Historically, the problem of MTSC was addressed by non-linear transforms of the time-series on which standard classification algorithms are used. Bostrom & Bagnall (2017) used a modification of the shapelet transform to quickly find k shapelets of length n to represent the time-series. They then use the kxn feature vector in a standard classification model. Similarly, BagOfPatterns (Lin et al., 2012) use the frequency of each word (token/feature) in the series to represent the series as a histogram of words. Bag of Symbolic Fourier Approximation Symbols (BOSS) (Large et al., 2018) uses Symbolic Fourier Approximation on frequencies of words in the series. Word ExtrAction for time SEries cLassification (WEASEL) (Schäfer & Leser, 2017) extracts words with multiple sliding windows of different sizes and selects the most discriminating words according to the chi-squared

[1] `https://github.com/lucidrains/vit-pytorch`

test. RandOm Convolutional KErnel Transform (ROCKET) (Dempster et al., 2019) generates a large number (10k) random convolutional kernels and extracts two features from the convolutions: the maximum and the proportion of positive values. It's originally designed for univariate time-series classification (UTSC) and later adapted to MTSC. Though ROCKET shows great performance on smaller MTS benchmark datasets, using it for datasets with hundreds of millions of MTS data is impractical (refer to 4.3).

DNN approaches for MTSC have borrowed heavily from image processing models - mainly Convolutional Neural Nets (CNN). Wang et al. (2016); Lin et al. (2020) used attention based CNNs, while Fawaz et al. (2019) used an ensemble of CNNs and Zou et al. (2019); Fawaz et al. (2018) integrated residual networks with CNNs. Similarly, Transformers were used by Yang et al. (2021) for UTSC. Rußwurm et al. (2019); Zerveas et al. (2021) used transformer for the MTSC, and we have compared and contrasted their approaches with our proposed approach (refer to 4.6).

3 Method

3.1 Vision Transformer for Multivariate Time-Series Classification (VitMTSC)

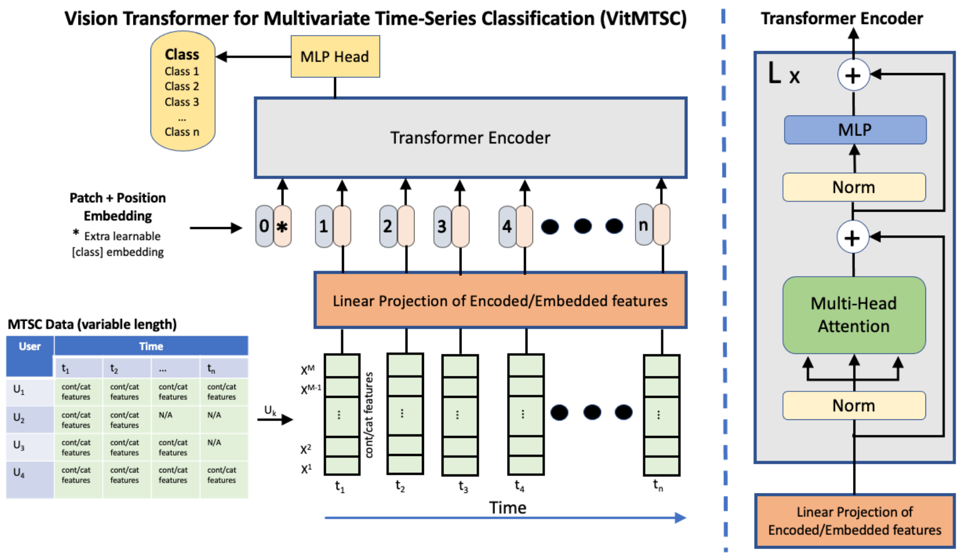
Vision Transformer for Multivariate Time-Series Classification (VitMTSC)
Class
Class 1
Class 2
Class 3
...
Class n
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 *
1
2
3
4
n
Linear Projection of Encoded/Embedded features
MTSC Data (variable length)
User
Time
t1
t2
...
tn
U1
U2
U3
U4
cont/cat features
N/A
Uk
X^M
X^{M-1}
X^2
X^1
cont/cat features
t1
t2
t3
t4
tn
Time
Transformer Encoder
L x
MLP
Norm
Multi-Head Attention
Norm
Linear Projection of Encoded/Embedded features

Figure 1: Vision Transformer for Multivariate Time-Series Classification (VitMTSC) - adapted from ViT (Dosovitskiy et al., 2020).

Figure 1 gives an overview of the VitMTSC architecture. We follow the original ViT design as closely as possible to take advantage of its scalable architecture, efficient implementations and future innovations (e.g., Bao et al. (2022)). Many variants of ViT has been proposed [2] but we will describe changes made to the original ViT model to handle MTSC.

3.1.1 Tokenization: How time-series data is encoded for transformer encoder?

The original Transformer architecture uses 1D sequence of token-ids as input, where a token-id is index of a word or sub-word in the vocabulary (V), and number of elements in this 1D sequence is the input sequence length. In the ViT, 2D images are first divided into multiple smaller 16x16 2D patches, and then each patch is flattened as 1D vector. This 1D vector is the input to the transformer, and total number of image patches serve as the effective input sequence length. Similar to the standard ViT, we start with MTS data, $\mathbf{X}$, and the number of time-steps in the MTS gives us input sequence length, N for the Transformer. However, though input to a ViT is strictly numerical (pixel values), an MTS datapoint can contain a combination of continuous and categorical features (Table 2). There are multiple ways to encode continuous and categorical features. Each individual feature of an MTS datapoint is separately encoded to produce corresponding 1D vector. Then these encoded 1D vectors

[2] `https://github.com/lucidrains/vit-pytorch`

are aggregated (concatenated) and presented in 1D format as input to the VitMTSC transformer encoder.

Feature Encoding. There are different ways of encoding features. An individual continuous numerical feature of an MTS datapoint can be represented as a real-valued scalar, as a linear projection from a scalar, or as a Soft One-Hot Encoding (SOHE) (Li et al., 2018; de Souza Pereira Moreira et al., 2021). Categorical features in real-world datasets can take on millions of distinct values. Thus, encoding like one-hot encoding that use vectors proportional to the cardinality of the feature values, do not work for commercial datasets. Instead we use target encoding (Rodríguez et al., 2018). For each categorical feature, the mean of a continuous target column is calculated, and the group-specific mean of each row is used to create a new feature (column). To prevent overfitting, the mean is computed via k-fold cross-validation and smoothed with the overall mean of the target variable. This transformation results in a single feature for each categorical variable; not increasing the dimensionality of the dataset. In our dataset, we normalized continuous numerical scalar features and target encoded categorical features (NVIDIA, 2022). This helped keep model complexity low, in comparison to using corresponding embeddings (e.g., Patel & Bhattacharyya (2017) shows that embedding dimension lower than 19 are not performant and our corresponding commercial embedding size is at least 256).

Padding and Truncation. Since time-series data can have variable length, we post-pad the input array with 0s to generate sequences of a pre-defined length N. Depending on the application, longer sequences are truncated to either the first or last N datapoints or use NLP style chunking (Briggs, 2021). We choose N to be the 99^{th} percentile length in the dataset in our real-world application(Table 2).

3.1.2 Token embedding

The original Transformer receives as input a 1D sequence of NLP tokens, map it to D dimensions with a trainable linear projection, and uses this constant latent vector size D through all of its layers. To handle 2D images, ViT, reshapes the original image in multiple smaller *patches*. They then flatten each patch and embed it as D-dimensional vector with a trainable linear projection. Similar to ViT, we embed the input sequence with a trainable linear projection ($\mathbf{E}$ in Eq. 1). The token embedding learns to represent each MTS datapoint as a vector in $\mathbb{R}^{d_{\mathrm{E}}}$. Similar to BERT's [`CLS`] and ViT's [`class`] token, VitMTSC prepends a learnable embedding to the sequence of embedded patches (time-steps) ($\mathbf{z}_0^0 = \mathbf{x}_{class}$), whose state at the output of the Transformer encoder serves as the representation of the time-series. In addition, to patch embeddings we also use positional embeddings ($\mathbf{E_{pos}}$). We discuss positional embeddings in Section 3.1.3.

$$\mathbf{z}_0 = [\mathbf{x}_{class}; \mathbf{X}_{t_0}\mathbf{E}; \mathbf{X}_{t_1}\mathbf{E}; ...; \mathbf{X}_{t_N}\mathbf{E}] + \mathbf{E}_{pos} \tag{1}$$

3.1.3 Positional embedding

Time-series data is ordered but Transformers are permutation invariant without positional information. The original Transformer (Vaswani et al., 2017) uses hand-crafted *absolute positional encoding*. Since the relative time between the data-points can be important for MTSC, we add *learnable 1D position embeddings* ($\mathbf{E_{pos}}$; see Algorithm 2 of Phuong & Hutter (2022)) to the time-step (patch) embeddings to retain this positional information before feeding input to Transformer encoder layer. The positional embedding is added to the token embedding to form a token's initial embedding for the transformer encoder (Eq. 1).

Positional Encoding as Timestamp Features Some datasets can have a set of (unordered) data points with the same timestamp. For example, for time-ordered set of products purchased by a customer, a single customer cart (same timestamp) may include multiple products. In such a case, absolute or learnable positional encodings will impose an arbitrary order on these datapoints (products). To avoid such an arbitrary order we do not use positional encoding on such datasets. Zhou et al. (2020) added both learnable and timestamp positional encoding to token embedding. Instead, we include the timestamp information (month, day, hour, etc of the cart purchase) of the datapoints as categorical features in the time-series. We found that the VitMTSC is able to use this information better than an arbitrary order imposed by the learnable positional encoding (refer Table 5).

3.1.4 Attention

Attention is the main architectural component of transformers; see Algorithm 3 of Phuong & Hutter (2022). In practice, transformers use multi-headed self-attention (MSA) – multiple attention heads (with separate learnable parameters) run in parallel – and combine their outputs; see Algorithm 5 of Phuong & Hutter (2022). VitMTSC also uses padding mask – it first converts 0s to 1s, and non-0 values to 0s, and then multiplies the padding mask with a large negative value. In VitMTSC's encoder MSA, the padding mask is added to the scaled score so that the original empty locations are now replaced with large negative values which will be eventually ignored by softmax activation function; see Algorithm 4 of Phuong & Hutter (2022), called with $\boldsymbol{Z} = \boldsymbol{X}$ and Mask.

3.1.5 Layer Normalization

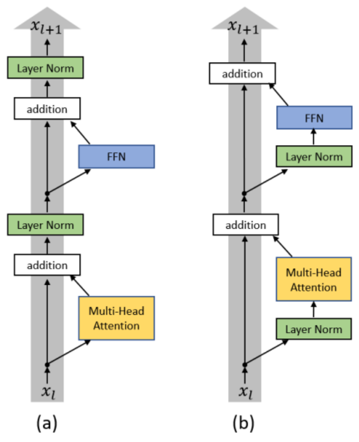

Figure 2: (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer. - adapted from Xiong et al. (2020).

Layer Normalization (Ba et al., 2016) explicitly controls the mean and variance of individual neural network activations; see Algorithm 4 of Phuong & Hutter (2022). The original Transformer uses Layer Normalization **after** "MSA/FFN block + Residual connection" in the transformer encoder – called *POSTNORM*. Recently, GPT-2 (Radford et al., 2019), ViT (Dosovitskiy et al., 2020) and the ViTMTSC, use Layer Normalization just **before** "MSA/FFN block" in the transformer encoder – called *PRENORM* (Figure 2).

3.1.6 Encoder-only transformer and model prediction

Like ViT, the VitMTSC transformer encoder consists of alternating layers of multiheaded self-attention (MSA) + residual connection and multilayer perceptron (MLP) + residual connection. The MLP contains two layers with a Gaussian Error Linear Unit (GELU) (Hendrycks & Gimpel, 2016) non-linearity. Layernorm (LN) is applied just before MSA/MLP block and then residual connections are applied (Figure 2). VitMTSC's [`class`] token state at the output of the Transformer encoder ($\mathbf{z}_L^0$) serves as the time-series representation $\mathbf{y}$ (Eq. 2). The classification head is implemented by a multi-layer perceptron (MLP) with one hidden layer and is attached to $\mathbf{y}$ — the [`class`] token output of the transformer encoder; see detailed Algorithm 1, which has been adapted from Algorithm 8 of Phuong & Hutter (2022) [3].

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0) \tag{2}$$

4 Experiments

We evaluate VitMTSC model along with current SOTA MTSC models on both publicly available and proprietary datasets.

[3]All current transformer models are some variation of the original Transformer. To set a new standard in DL publishing, Phuong & Hutter (2022) authors explicitly encourage the reader to copy and adapt them to their needs and cite the original as "adapted from [PH22]".

Algorithm 1: $\boldsymbol{P} \leftarrow$ ETransformer($\boldsymbol{x}|\theta$) - adapted Algorithm 8 from Phuong & Hutter (2022)

```
Input: $x \in V^*$, an MTS sequence. Let $V$ denote a set, such that $\mathbf{X}_{t_K} \in V$, often identified with $[N_V] := \{1, ..., N_V\}$. Let $\mathbf{X} \equiv x \equiv x[1:\ell] \equiv x[1]x[2]...x[\ell] \in V^*$.
Output: $p \in \Delta(C)$, a probability distribution over the classes in task.
Hyperparameters: $\ell_{max}, L, H, d_e, d_{mlp}, d_f \in \mathbb{N}$
Parameters: $\theta$ includes all of the following parameters:
$W_e \in \mathbb{R}^{d_e \times N_V}$, $W_p \in \mathbb{R}^{d_e \times \ell_{max}}$, the token embedding and positional embedding matrices.
For $l \in [L]$ :
    $W_l$, multi-head attention parameters for the layer $l$,
    $\gamma_l^1, \beta_l^1, \gamma_l^2, \beta_l^2 \in \mathbb{R}^{d_e}$, two sets of layer-norm parameters,
    $W_{mlp1}^l \in \mathbb{R}^{d_{mlp} \times d_e}$, $b_{mlp1}^l \in \mathbb{R}^{d_{mlp}}$, $W_{mlp2}^l \in \mathbb{R}^{d_{mlp} \times d_e}$, $b_{mlp2}^l \in \mathbb{R}^{d_{mlp}}$, MLP parameters.
$W_{mlp3}^{out1} \in \mathbb{R}^{d_e \times d_{mlp}}$, $b_{mlp3}^{out1} \in \mathbb{R}^{d_{mlp}}$, $W_{mlp4}^{out2} \in \mathbb{R}^{d_e \times d_{mlp}}$, $b_{mlp4}^{out2} \in \mathbb{R}^{d_{mlp}}$, the final linear projection and layer-norm parameters.
1   $\ell \leftarrow$ length($x$)
2   for $t \in [\ell]$ : $e_t \leftarrow W_e[:, x[t]] + W_p[:, t]$
3   $X \leftarrow [e_1, e_2, ...e_l]$
4   for $l = 1, 2, ..., L$ do
5       for $t \in [\ell]$ : $\tilde{X}[:, t] \leftarrow$ layer_norm($X[:, t], \gamma_l^1, \beta_l^1$)
6       $X \leftarrow X+$ MHAttention ($\tilde{X}$ | $W_l$, Mask $\equiv 1$)
7       for $t \in [\ell]$ : $\tilde{X}[:, t] \leftarrow$ layer_norm($X[:, t], \gamma_l^2, \beta_l^2$)
8       $X \leftarrow X + W_{mlp2}^l$ GELU($W_{mlp1}^l \tilde{X} + b_{mlp1}^l \mathbf{1}^\mathbf{T}$) $+ b_{mlp2}^l \mathbf{1}^\mathbf{T}$
9   end
10  $X = X[:, 1]$, discard everything and keep only [class] token.
11  for $t \in [\ell]$ : $X[:, t] \leftarrow$ layer_norm($X[:, t], \gamma, \beta$)
12  $X \leftarrow X + W_{mlp4}^{out2}$ GELU($W_{mlp3}^{out1} X + b_{mlp3}^{out1} \mathbf{1}^\mathbf{T}$) $+ b_{mlp4}^{out2} \mathbf{1}^\mathbf{T}$
13  return $p$ = softmax($X$)
```

Table 1: Open-source UEA datasets used for evaluation. It has only continuous (CONT) features and time-series datapoints are combined across both train and test datasets.

DATASET	LENGTH	CONT	CAT	CLASSES	TRAIN	TEST	TS DATAPOINTS
PenDigits	8	2	N/A	10	7,494	3,498	87,936
SpokenArabicDigits	4-93	13	N/A	10	6,599	2,199	350,253
CharacterTrajectories	60-182	3	N/A	20	1,422	1,436	342,939
FaceDetection	62	4	N/A	6	5,890	3,524	583,668
InsectWingbeat	2-22	200	N/A	10	25,000	25,000	335,534

4.1 Data

The majority of algorithms in the MTSC literature have been evaluated on open-source UEA datasets (Ruiz et al., 2021). The individual datasets (Table 1) in the UEA datasets are very small (the largest dataset has only 50k time-series data). Modern commercial datasets on the other hand contain hundreds of millions of time-series datapoints (refer Table 2). Also, UEA datasets only contain continuous variables, whereas real-world datasets can include a mix of continuous, discrete, and categorical variables. Though we evaluate VitMTSC on UEA datasets, the real power of the method is seen in real-world data.

4.1.1 Open-source UEA Datasets

The UEA MTSC archive is a repository of benchmark datasets for time-series classification (Bagnall et al., 2018). It includes 30 multivariate datasets, four of which have variable lengths. We test our model on the five datasets (refer Table 1) available in the UEA repository, which include three datasets of variable length, a condition frequently observed in real-world data.

4.1.2 Real-World Commercial Datasets

We used VitMTSC and other DNN MTSC models to analyze clickstream data on 9 different stores of a popular e-retailer. The aim of our study was to identify the set of customers that would be responsive

Table 2: Commercial datasets with both continuous (CONT) and categorical (CAT) features

DATASET	LENGTH	CONT	CAT	CLASSES	TRAIN	VALID	TEST	TS DATAPOINTS
Small	5-300	8	20	2	595K	198K	198K	94M
Large	10-300	8	20	2	4.2M	0.9M	0.9M	922M

to specific marketing campaigns. Using click-stream data as input, we build a binary classifier with the target variable being respond/not respond to a marketing campaign. We use response data from similar historical marketing campaigns for each retail store and train a separate model for each store. One of our smallest store has 992k time-series while the largest one has 6m. Each data point in the time-series consists of 20 features – 8 continuous and 12 categorical – that include products that the customer has clicked/purchased, product category, interaction type (click/purchase), time features (month, day, hour, etc), product attributes (price, discount, etc), customer attributes and more. Also, the space of values that a categorical variable can take can be quite large (e.g.: product ID can be one of hundreds of thousands to hundreds of millions of products depending on store).

4.2 Results

4.3 Benchmark Datasets

For MTSC, ROCKET (Dempster et al., 2019), and CMFMTS (Baldan & Benítez, 2020) are current SOTA Non-DNN methods and TST (Zerveas et al., 2021) is current SOTA DNN method. On the datasets of interest to us, they outperform all other models. Since UEA datasets specify both train and test datasets, we are able to use performance numbers from Baldan & Benítez (2020) and Zerveas et al. (2021) as CMFMTS and TST performances respectively. Unlike current practice in MTSC research(Section 6.6 of Fawaz et al. (2019), and Section 4 of Zerveas et al. (2021)), we further divided train data in 80-20% split and use this new validation dataset to tune both hyper-parameters and guide training to avoid overfitting on test dataset. We use provided test dataset only in the end to check model generalization capabilities, like we do in commercial applications. Table 3 gives the average accuracy of CMFMTS, TST, ROCKET and VitMTSC on these five open-source MTSC datasets.

ROCKET is one of the most prominent time-series classification algorithm. Unlike Zerveas et al. (2021), we were able to train ROCKET on all open-source datasets using sktime implementation[4]. However, on a p3dn.24xlarge (768 GB RAM and 96 CPUs) machine, ROCKET took 2 hours 40 minutes to complete training/inference on the biggest open-source dataset, InsectWingbeat (Table 1) that has only 50K time-series datapoints. This huge run-time renders it unusable in commercial datasets (Table 2) with millions of time-series datapoints and we won't discuss it further.

Table 3: Average accuracy of VitMTSC and other models on UEA datasets. A dash indicates that the corresponding dataset was not evaluated in the respective paper. The **VitMTSC(val)** column shows model overfit, since it was also used for hyper-parameter tuning.

DATASET	CMFMTS	TST	ROCKET	VitMTSC(val)	VitMTSC(test)
PenDigits	95.9	-	97.8	**99.4**	97.2
SpokenArabicDigits	97.5	**99.8**	**99.8**	99.0	96.9
CharacterTrajectories	97.1	-	**99.3**	97.1	98.5
FaceDetection	57.9	68.9	64.0	**72.7**	67.0
InsectWingbeat	67.7	**68.7**	53.6	60.4	60.3

As we can see, model overfits validation set since it was used for both hyper-parameter tuning and guide training but generalized poorly on 3 out of 5 unseen test datasets - especially on smaller datasets. We hope that future MTSC papers will follow our approach and won't use provided test dataset for

[4]`https://www.sktime.org/en/latest/index.html`

either training or hyper-parameter tuning. This will help benchmark generalization capabilities of MTSC models correctly. Since Zerveas et al. (2021) has already explored impact of pre-training on UEA datasets, we didn't. Rather, in future, we would like to explore pre-training methods that works well on ViT e.g., BEiT (Bao et al., 2022).

4.4 Offline Experiments on Commercial Datasets

We compared our model against several DNN time-series classifiers that learns, like VitMTSC, end-to-end directly from raw MTS data without feature engineering. For comparative analysis, we implemented and evaluated six published MTSC DL models — FCN (Wang et al., 2016), Inception Time (Fawaz et al., 2019), ResCNN (Zou et al., 2019), ResNet (Fawaz et al., 2018), CropTransformer (Rußwurm et al., 2019) and TST (Zerveas et al., 2021) — and the VitMTSC model on our smallest dataset (refer Table 2). We divided this dataset into 595K (336K +ve and 259K -ve) training, 198K (112K +ve and 86K -ve) validation and 198K (112K +ve and 86K -ve) test sets. Our time-series sample consisted of 10-300 datapoints depending on user activities. Since our data is unbalanced, we use class weights to account for the imbalance. Also, as mentioned in Section 3.1.3, we do not use absolute or learnable positional encodings and instead include the temporal information in our multi-variate data - timestamp encoding. Table 4 shows the PRAUC for the different models. Based on these results, we trained only our VitMTSC model for our larger datasets with 6M datapoints.

Table 4: PRAUC of VitMTSC vs different DL models on data from one of our smaller stores. The model predicts whether a customer will respond/not respond to a marketing campaign.

MODEL	**RESPOND**	**NOT RESPOND**
FCN	0.57	0.47
Inception Time	0.56	0.42
ResCNN	0.55	0.41
ResNet	0.54	0.44
CropTransformer	0.55	0.45
TST	0.56	0.45
VitMTSC	**0.96**	**0.95**

4.5 Online Experiment on Commercial Datasets

In an online A/B test between VitMTSC and an existing heuristic-based baseline method for finding potential customers who will respond to a marketing campaign, the conversion rate (the percentage of customers who responded to the campaign out of those who were targeted) went up by an average of 352% (p-value < 0.001).

4.5.1 Ablation Study on Commercial Datasets

We performed four ablations to evaluate the importance of placement of layer-normalization, padding masks, class weights and positional encoding (refer Table 5). Though we performed the ablations on nine datasets, for simplicity, we present results from only two datasets (refer Table 2) with the understanding that other datasets followed similar trends. Section 3.1.5 and Section 4.6 provides detailed information on Layer-Normalization and its impact. As also noticed by Nguyen & Salazar (2019), even with learning rate warmup, the VitMTSC PostNorm version failed to converge on our datasets. Adding explicit positional encoding deteriorates performance. This is because positional encoding adds an arbitrary order to multiple activities that occurred at the same time. Instead using the categorical time features (month, day, hour, etc) improves the performance of the model. The importance of class-weight depends on the amount of data. Only in the Large dataset where the data size is an order of magnitude more than other datasets, class-weighting was not required. All other datasets (eight stores - though Table 5 report results on only one of the eight stores and the largest store) showed improved performance with class weights. Similarly, masking allowed the classifier to ignore irrelevant (padded) data-points and improved performance. Table 5 gives the average PRAUC of the VitMTSC with different configurations on these real-world commercial datasets.

Table 5: VitMTSC ablation study: average PRAUC on two commercial datasets. We test out (a) Pre and Post layer-normalization, (b) using learnable positional encoding vs timestamp positional encoding (including the temporal information as features in our multi-variate data), (c) using class weights and (d) padding masks. Each ablation uses the best values from the other three combinations.

DATASET	LAYER-NORM		POSITION-ENCODING		CLASS-WEIGHT		MASK	
	PRE	POST	LEARNABLE	TIMESTAMP	YES	NO	YES	NO
Small	**0.961**	0.574	0.943	**0.961**	**0.961**	0.955	**0.961**	0.957
Large	**0.975**	0.528	0.949	0.971	0.971	**0.975**	0.971	0.969

4.6 Comparison of Other MTSC Transformer Models to VitMTSC

CropTransformer (Rußwurm et al., 2019) and TST (Zerveas et al., 2021) are other two Transformer models for MTSC. The CropTransformer official implementation uses Rectified Linear Unit (RELU) (Nair & Hinton, 2010) activation function, *POSTNORM* layer-normalization (Ba et al., 2016), and hand-crafted absolute positional encoding like the original Transformer (Vaswani et al., 2017). TST uses *POSTNORM* like the original transformer but replaces layer-normalization with batch-normalization (Ioffe & Szegedy, 2015), learnable positional encoding and GELU activation function. VitMTSC on the other hand, uses *PRENORM* layer-normalization, GELU activation function and learnable positional and timestamp encoding (refer to 3.1.3). Also, similar to BERT's [`CLS`] and ViT's [`class`] token, VitMTSC uses only [`class`] token at output layer for MLP and discards rest of transformer encoder output. In contrast, both CropTransformer and TST don't use [`CLS`] token and feed full output of transformer encoder, in the end, to the MLP layer for classification. This [`class`] token provides an aggregate representation of MTS sequence to the final MLP Head for classification. Table 4, and Table 5 show impact of these choices in model performance on commercial datasets (4.1.2). Both official implementations of TST[5] and CropTransformer[6] are in PyTorch and PyTorch offical transformer encoder implementation added support for PreNorm(default=False) in mid-2021[7]. For this reason, we decided to provide the Algorithm for VitMTSC (3.1.6).

The main reason why VitMTSC significantly outperform CropTransformer (Rußwurm et al., 2019) and TST (Zerveas et al., 2021) on real-world datasets is: VitMTSC, similar to ViT, uses *PRENORM* layer-normalization(refer to 3.1.5). Nguyen & Salazar (2019); Xiong et al. (2020) provide detailed insights of impact of *PRENORM* vs *POSTNORM* in Transformer. In our ablation study on using *PRENORM* and *POSTNORM* (refer Table 5), we found that *PRENORM* is the key to learning on larger commercial datasets (refer Table 2). As mentioned in Section 4.5.1 and also noticed by Nguyen & Salazar (2019), even with learning rate warmup, the VitMTSC PostNorm version failed to converge on our datasets.

5 Conclusion and future work

We introduce Vision Transformer for Multivariate Time-Series Classification (VitMTSC), a novel deep learning architecture for multivariate time-series classification (MTSC) based on ViT (Dosovitskiy et al., 2020). We demonstrate, on open-source datasets, VitMTSC's performance matches the current SOTA methods, the model scales to large datasets seen in commercial retail sites and provides far superior performance than other neural net based MTSC models. Since, the VitMTSC model can learn directly from raw multivariate time-series data, we can avoid the manual feature engineering, which is a fundamental part of non-DNN MTSC algorithms.

Finally, we emphasize the significance of *PRENORM* Layer normalization in transformer networks. Attempts have been made to comprehend ViT's success (Raghu et al., 2021; Trockman & Kolter, 2022) and enhance its performance further (Chen et al., 2021). As part of further research, we will investigate effect of *PRENORM/POSTNORM* Layer normalization on ViT, since we observed significant impact in our commercial MTS datasets.

[5]`https://github.com/gzerveas/mvts_transformer`

[6]`https://github.com/dl4sits/BreizhCrops`

[7]`https://github.com/pytorch/pytorch/issues/55270`

REFERENCES

Fevzi Alimoglu. Combining multiple classifiers for pen-based handwritten digit recognition, 1996. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.6299&rep=rep1&type=pdf.

Sercan Ömer Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. *CoRR*, abs/1908.07442, 2019. URL http://arxiv.org/abs/1908.07442.

Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lucic, and Cordelia Schmid. Vivit: A video vision transformer. *CoRR*, abs/2103.15691, 2021. URL https://arxiv.org/abs/2103.15691.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Anthony J. Bagnall, Hoang Anh Dau, Jason Lines, Michael Flynn, James Large, Aaron Bostrom, Paul Southam, and Eamonn J. Keogh. The UEA multivariate time series classification archive, 2018. *CoRR*, abs/1811.00075, 2018. URL http://arxiv.org/abs/1811.00075.

Francisco Javier Baldan and José Manuel Benítez. Multivariable times series classification through an interpretable representation. *CoRR*, abs/2009.03614, 2020. URL https://arxiv.org/abs/2009.03614.

Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. BEit: BERT pre-training of image transformers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=p-BhZSz59o4.

Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V. Le. Attention augmented convolutional networks. *CoRR*, abs/1904.09925, 2019. URL http://arxiv.org/abs/1904.09925.

Konstantinos Benidis, Syama Sundar Rangapuram, Valentin Flunkert, Bernie Wang, Danielle C. Maddix, Ali Caner Türkmen, Jan Gasthaus, Michael Bohlke-Schneider, David Salinas, Lorenzo Stella, Laurent Callot, and Tim Januschowski. Neural forecasting: Introduction and literature overview. *CoRR*, abs/2004.10240, 2020. URL https://arxiv.org/abs/2004.10240.

Ane Blázquez-García, Angel Conde, Usue Mori, and José Antonio Lozano. A review on outlier/anomaly detection in time series data. *CoRR*, abs/2002.04236, 2020. URL https://arxiv.org/abs/2002.04236.

Aaron Bostrom and Anthony J. Bagnall. A shapelet transform for multivariate time series classification. *CoRR*, abs/1712.06428, 2017. URL http://arxiv.org/abs/1712.06428.

James Briggs. *How to Apply Transformers to Any Length of Text*, 2021. URL https://towardsdatascience.com/how-to-apply-transformers-to-any-length-of-text-a5601410af7f.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *CoRR*, abs/2005.12872, 2020. URL https://arxiv.org/abs/2005.12872.

Hila Chefer, Shir Gur, and Lior Wolf. Transformer interpretability beyond attention visualization. *CoRR*, abs/2012.09838, 2020. URL https://arxiv.org/abs/2012.09838.

Kaixuan Chen, Dalin Zhang, Lina Yao, Bin Guo, Zhiwen Yu, and Yunhao Liu. Deep learning for sensor-based human activity recognition: Overview, challenges and opportunities. *CoRR*, abs/2001.07416, 2020. URL https://arxiv.org/abs/2001.07416.

Xinlei Chen, Saining Xie, and Kaiming He. An empirical study of training self-supervised vision transformers. *CoRR*, abs/2104.02057, 2021. URL https://arxiv.org/abs/2104.02057.

Yanping Chen and et al. Adena Why. Ucr computational entomology, 2017. URL https://www.cs.ucr.edu/~eamonn/CE/.

Yanping Chen, Adena Why, Gustavo E. A. P. A. Batista, Agenor Mafra-Neto, and Eamonn J. Keogh. Flying insect classification with inexpensive sensors. *CoRR*, abs/1403.2654, 2014. URL http://arxiv.org/abs/1403.2654.

Kukjin Choi, Jihun Yi, Changhwa Park, and Sungroh Yoon. Deep learning for anomaly detection in time-series data: Review, analysis, and guidelines. *IEEE Access*, 9:120043–120065, 2021. doi: 10.1109/ACCESS.2021.3107975.

Conda. *Conda: open source package management system and environment management system*, 2020. URL https://docs.conda.io/en/latest/.

Zihang Dai, Zhilin Yang, Fan Yang, William W. Cohen, and Ruslan Salakhutdinov. Good semi-supervised learning that requires a bad GAN. *CoRR*, abs/1705.09783, 2017. URL http://arxiv.org/abs/1705.09783.

Gabriel de Souza Pereira Moreira, Sara Rabhi, Jeong Min Lee, Ronay Ak, and Even Oldridge. Transformers4rec: Bridging the gap between nlp and sequential / session-based recommendation. In *Proceedings of the 15th ACM Conference on Recommender Systems*, RecSys '21, pp. 143–153, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384582. doi: 10.1145/3460231.3474255. URL https://doi.org/10.1145/3460231.3474255.

Angus Dempster, François Petitjean, and Geoffrey I. Webb. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *CoRR*, abs/1910.13051, 2019. URL http://arxiv.org/abs/1910.13051.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL https://arxiv.org/abs/2010.11929.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

fast.ai. nbdev - create delightful python projects using jupyter notebooks. Github, 2020. URL https://github.com/fastai/nbdev.

Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *CoRR*, abs/1809.04356, 2018. URL http://arxiv.org/abs/1809.04356.

Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F. Schmidt, Jonathan Weber, Geoffrey I. Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. Inceptiontime: Finding alexnet for time series classification. *CoRR*, abs/1909.04939, 2019. URL http://arxiv.org/abs/1909.04939.

Arash Gharehbaghi, Per Ask, and Ankica Babic. A pattern recognition framework for detecting dynamic changes on cyclic time series. *Pattern Recognition*, 48(3):696–708, 2015. ISSN 0031-3203. doi: 10.1016/j.patcog.2014.08.017. URL https://www.sciencedirect.com/science/article/pii/S0031320314003318.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Nacereddine Hammami and Mokhtar Sellam. Tree distribution classifier for automatic spoken arabic digit recognition. In *2009 International Conference for Internet Technology and Secured Transactions, (ICITST)*, pp. 1–4, Nov 2009. doi: 10.1109/ICITST.2009.5402575.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017. URL `http://arxiv.org/abs/1712.00409`.

Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar S. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *CoRR*, abs/2012.06678, 2020. URL `https://arxiv.org/abs/2012.06678`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Yohan Jo, Keith Maki, and Gaurav Tomar. Time series analysis of clickstream logs from online courses. *CoRR*, abs/1809.04177, 2018. URL `http://arxiv.org/abs/1809.04177`.

Kaggle. Decmeg2014 - decoding the human brain - predict visual stimuli from meg recordings of human brain activity, 2017. URL `https://www.kaggle.com/competitions/decoding-the-human-brain/data`.

Juha Karhunen, Tapani Raiko, and KyungHyun Cho. Chapter 7 - unsupervised deep learning: A short review. In Ella Bingham, Samuel Kaski, Jorma Laaksonen, and Jouko Lampinen (eds.), *Advances in Independent Component Analysis and Learning Machines*, pp. 125–142. Academic Press, 2015. ISBN 978-0-12-802806-3. doi: https://doi.org/10.1016/B978-0-12-802806-3.00007-5. URL `https://www.sciencedirect.com/science/article/pii/B9780128028063000075`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

James Large, Anthony J. Bagnall, Simon Malinowski, and Romain Tavenard. From BOP to BOSS and beyond: Time series classification with dictionary based classifiers. *CoRR*, abs/1809.06751, 2018. URL `http://arxiv.org/abs/1809.06751`.

Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *CoRR*, abs/1907.00235, 2019. URL `http://arxiv.org/abs/1907.00235`.

Yang Li, Nan Du, and Samy Bengio. Time-dependent representation for neural event sequence prediction. 2018. URL `https://openreview.net/pdf?id=HyrT5Hkvf`.

Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021. ISSN 2162-2388. doi: 10.1109/TNNLS.2021.3084827.

PyTorch Lightning. *PyTorch Lightning*, 2021. URL `https://github.com/Lightning-AI/lightning`.

Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194): 20200209, feb 2021. doi: 10.1098/rsta.2020.0209. URL `https://doi.org/10.1098%2Frsta.2020.0209`.

Huiwei Lin, Yunming Ye, Ka-Cheong Leung, and Bowen Zhang. A multivariate time series classification method based on self-attention. In Jeng-Shyang Pan, Jerry Chun-Wei Lin, Yongquan Liang, and Shu-Chuan Chu (eds.), *Genetic and Evolutionary Computing*, pp. 491–499, Singapore, 2020. Springer Singapore. ISBN 978-981-15-3308-2.

Jessica Lin, Rohan Khade, and Yuan Li. Rotation-invariant similarity in time series using bag-of-patterns representation. *J. Intell. Inf. Syst.*, 39(2):287–315, oct 2012. ISSN 0925-9902. doi: 10.1007/s10844-012-0196-5. URL `https://doi.org/10.1007/s10844-012-0196-5`.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *CoRR*, abs/2106.04554, 2021. URL https://arxiv.org/abs/2106.04554.

Markus Löning, Anthony J. Bagnall, Sajaysurya Ganesh, Viktor Kazakov, Jason Lines, and Franz J. Király. sktime: A unified interface for machine learning with time series. *CoRR*, abs/1909.07872, 2019.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. *CoRR*, abs/1910.05895, 2019. URL http://arxiv.org/abs/1910.05895.

NVIDIA. *RAPIDSAI: Collection of Libraries for End to End GPU Data Science*, 2018. URL https://rapids.ai.

NVIDIA. *NVTabular: GPU Accelerated ETL for Recommender Systems*, 2021. URL https://github.com/NVIDIA-Merlin/NVTabular.

NVIDIA. *NVTabular: GPU Accelerated ETL for Recommender Systems - Target Encoding*, 2022. URL https://nvidia-merlin.github.io/NVTabular/main/generated/nvtabular.ops.TargetEncoding.html.

Tin Lay Nwe, Tran Huy Dat, and Bin Ma. Convolutional neural network with multi-task learning scheme for acoustic scene classification. In *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 1347–1350, 2017. doi: 10.1109/APSIPA.2017.8282241.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, and Alexander Ku. Image transformer. *CoRR*, abs/1802.05751, 2018. URL http://arxiv.org/abs/1802.05751.

Kevin Patel and Pushpak Bhattacharyya. Towards lower bounds on number of dimensions for word embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 31–36, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing. URL https://aclanthology.org/I17-2006.

Uber Petastorm. *Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning*, 2021. URL https://github.com/uber/petastorm.

Mary Phuong and Marcus Hutter. Formal algorithms for transformers. 2022. doi: 10.48550/ARXIV.2207.09238. URL https://arxiv.org/abs/2207.09238.

Ploomber. Ploomber- we develop tools to streamline the development to production data science workflow. Ploomber, 2021. URL https://github.com/ploomber.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015. URL https://arxiv.org/abs/1511.06434.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019. URL https://openai.com/blog/better-language-models/.

Maithra Raghu, Thomas Unterthiner, Simon Kornblith, Chiyuan Zhang, and Alexey Dosovitskiy. Do vision transformers see like convolutional neural networks? *CoRR*, abs/2108.08810, 2021. URL https://arxiv.org/abs/2108.08810.

Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. Stand-alone self-attention in vision models. *CoRR*, abs/1906.05909, 2019. URL http://arxiv.org/abs/1906.05909.

Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *CoRR*, abs/2002.04803, 2020. URL `https://arxiv.org/abs/2002.04803`.

Pau Rodríguez, Miguel A. Bautista, Jordi Gonzàlez, and Sergio Escalera. Beyond one-hot encoding: Lower dimensional target embedding. *Image and Vision Computing*, 75:21–31, 2018. ISSN 0262-8856. doi: https://doi.org/10.1016/j.imavis.2018.04.004. URL `https://www.sciencedirect.com/science/article/pii/S0262885618300623`.

Alejandro Pasos Ruiz, Michael Flynn, James Large, Matthew Middlehurst, and Anthony Bagnall. The great multivariate time series classification bake off: A review and experimental evaluation of recent algorithmic advances. *Data Min. Knowl. Discov.*, 35(2):401–449, mar 2021. ISSN 1384-5810. doi: 10.1007/s10618-020-00727-3. URL `https://doi.org/10.1007/s10618-020-00727-3`.

Marc Rußwurm, Sébastien Lefèvre, and Marco Körner. Breizhcrops: A satellite time series dataset for crop type identification. *CoRR*, abs/1905.11893, 2019. URL `http://arxiv.org/abs/1905.11893`.

Patrick Schäfer and Ulf Leser. Fast and accurate time series classification with WEASEL. *CoRR*, abs/1701.07681, 2017. URL `http://arxiv.org/abs/1701.07681`.

José F. Torres, Dalil Hadjout, Abderrazak Sebaa, Francisco Martínez-Álvarez, and Alicia Troncoso. Deep learning for time series forecasting: A survey. *Big Data*, 9(1):3–21, 2021. doi: 10.1089/big.2020.0159. URL `https://doi.org/10.1089/big.2020.0159`. PMID: 33275484.

Asher Trockman and J Zico Kolter. Patches are all you need?, 2022. URL `https://openreview.net/forum?id=TVHS5Y4dNvM`.

Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. Tranad: Deep transformer networks for anomaly detection in multivariate time series data. *CoRR*, abs/2201.07284, 2022. URL `https://arxiv.org/abs/2201.07284`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

Yuqing Wan and Yain-Whar Si. A formal approach to chart patterns classification in financial time series. *Information Sciences*, 411:151–175, 2017. ISSN 0020-0255. doi: 10.1016/j.ins.2017.05.028. URL `https://www.sciencedirect.com/science/article/pii/S002002551631636X`.

Xiaolong Wang, Ross B. Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. *CoRR*, abs/1711.07971, 2017. URL `http://arxiv.org/abs/1711.07971`.

Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. *CoRR*, abs/1611.06455, 2016. URL `http://arxiv.org/abs/1611.06455`.

Qingsong Wen, Liang Sun, Fan Yang, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. Time series data augmentation for deep learning: A survey. In Zhi-Hua Zhou (ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pp. 4653–4660. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/631. URL `https://doi.org/10.24963/ijcai.2021/631`. Survey Track.

Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. Transformers in time series: A survey, 2022. URL `https://arxiv.org/abs/2202.07125`.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. *CoRR*, abs/2002.04745, 2020. URL `https://arxiv.org/abs/2002.04745`.

Jiehui Xu, Haixu Wu, Jianmin Wang, and Mingsheng Long. Anomaly transformer: Time series anomaly detection with association discrepancy. *CoRR*, abs/2110.02642, 2021. URL `https://arxiv.org/abs/2110.02642`.

Chao-Han Huck Yang, Yun-Yun Tsai, and Pin-Yu Chen. Voice2series: Reprogramming acoustic models for time series classification. *CoRR*, abs/2106.09296, 2021. URL `https://arxiv.org/abs/2106.09296`.

Zhaosheng Yang, Qichun Bing, Ciyun Lin, Nan Yang, and Duo Mei. Research on Short-Term Traffic Flow Prediction Method Based on Similarity Search of Time Series. *Mathematical Problems in Engineering*, 2014:1–8, August 2014. doi: 10.1155/2014/184632. URL `https://ideas.repec.org/a/hin/jnlmpe/184632.html`.

George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery amp; Data Mining*, KDD '21, pp. 2114–2124, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467401. URL `https://doi.org/10.1145/3447548.3467401`.

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. *CoRR*, abs/2012.07436, 2020. URL `https://arxiv.org/abs/2012.07436`.

Tian Zhou, Ziqing Ma, Qingsong Wen, Xue Wang, Liang Sun, and Rong Jin. Fedformer: Frequency enhanced decomposed transformer for long-term series forecasting. *CoRR*, abs/2201.12740, 2022. URL `https://arxiv.org/abs/2201.12740`.

Xiaowu Zou, Zidong Wang, Qi Li, and Weiguo Sheng. Integration of residual network and convolutional neural network along with various activation functions and global pooling for time series classification. *Neurocomputing*, 367:39–45, 2019. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.08.023. URL `https://www.sciencedirect.com/science/article/pii/S0925231219311506`.

A Appendix

A.1 Experiment Details

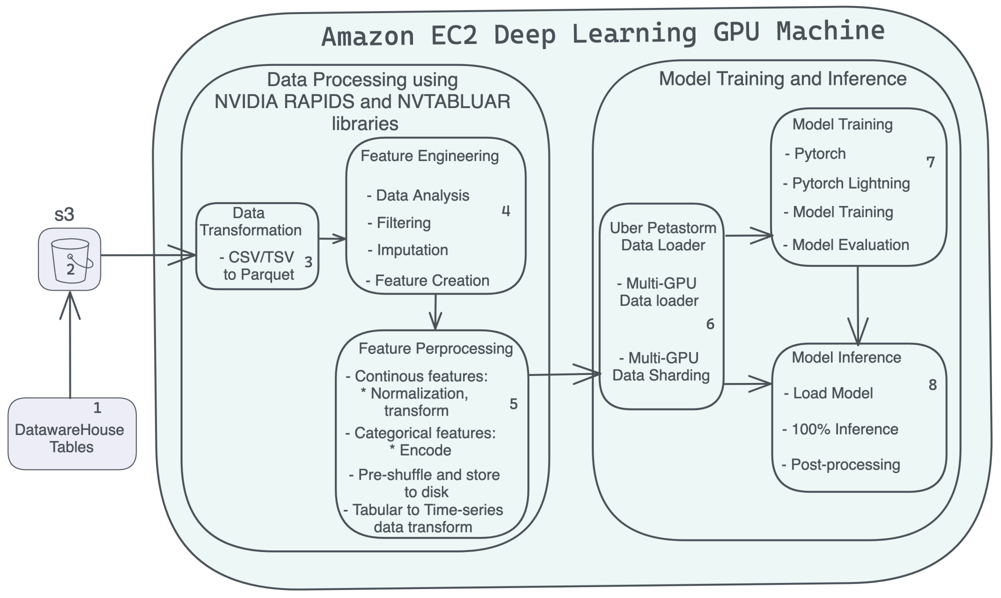
Amazon EC2 Deep Learning GPU Machine
Data Processing using NVIDIA RAPIDS and NVTABLUAR libraries
Model Training and Inference
s3
2
1
DatawareHouse Tables
Data Transformation
- CSV/TSV to Parquet
3
Feature Engineering
- Data Analysis
- Filtering
- Imputation
- Feature Creation
4
Feature Perprocessing
- Continous features: * Normalization, transform
- Categorical features: * Encode
- Pre-shuffle and store to disk
- Tabular to Time-series data transform
5
Uber Petastorm Data Loader
- Multi-GPU Data loader
- Multi-GPU Data Sharding
6
Model Training
- Pytorch
- Pytorch Lightning
- Model Training
- Model Evaluation
7
Model Inference
- Load Model
- 100% Inference
- Post-processing
8

Figure 3: VitMTSC deep learning training and inference pipeline

A.1.1 Tools

To efficiently process terabytes (TB) of commercial MTSC data, we use Graphical Processing Units (GPUs) for end-to-end to accelerated data processing as well as model training/inference (Raschka et al., 2020). We use a combination of open-source libraries: a) NVIDIA RAPIDS and NVTabular (NVIDIA, 2018; 2021) GPU accelerated libraries for data-processing and feature engineering; b) Uber Petastorm (Petastorm, 2021) based data-loaders for multi-gpu training/inference; c) Pytorch-Lightning (Lightning, 2021) for model training/inference; d) nbdev (fast.ai, 2020) to enable code sharing across Jupyter notebooks; and e) Conda (Conda, 2020) and Ploomber (Ploomber, 2021) for workflow automation. With help of these tools, we are able to write end-to-end ML pipeline in Jupyter and test it locally or deploy it to Cloud. To reproduce our experiment results on open-source UEA datasets, we provide the fully self-contained code in supplemental material[8].

A.1.2 Environment

We run all experiments locally on Amazon Elastic Compute Cloud (EC2) p3dn.24xlarge machine or deployed full workflow remotely on Amazon Web Service (AWS) Batch service on g4dn.4xlarge machines. All machines have AWS Deep Learning Amazon Machine Images (DLAMI) and Ubuntu operating system. The p3dn.24xlarge machine has 96 CPUs, 768GB RAM and 256GB of total GPU memory divided across eight GPU cores. Similarly, g4dn.4xlarge machine has 16 CPUs, 64GB RAM and 16GB of total GPU memory on a single GPU core.

A.1.3 Techniques

The open-source UEA datasets (refer Table 1) used a single NVIDIA Tesla V100 GPU core whereas commercial datasets (refer Table 5) used all eight NVIDIA Tesla V100 GPU cores for both training and inference. We use the Extract step of Extract-Transform-Load (ETL) to pull data from Dataware House (DW) to Amazon Simple Storage Service (Amazon S3) and use GPU workflow (Figure 3) for the remaining steps of the ML modeling.

[8]To facilitate further research, we release code at `https://github.com/mtsc-research/vitmtsc`.

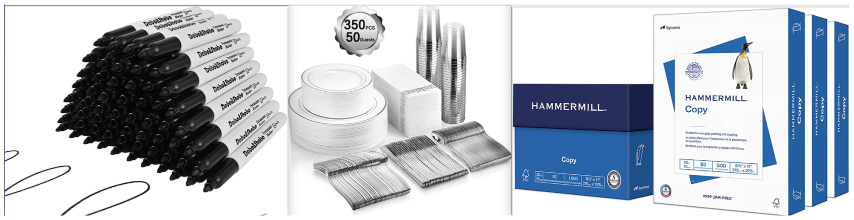

Figure 4: Top 3 products - all business-related bulk purchases - identified by VitMTSC model indicate that the user will be responsive to a marketing campaign targeted towards small businesses.

As shown in Figure 3, we keep data in tabular format until last stage of **Step 5** for data-processing and feature engineering. And then we convert tabular data to time-series format, such that, all data related to a single time-series are in a single row in parquet file. This helps Uber Petastorm (Petastorm, 2021) based data-loaders to load all data for a time-series in one read during multi-gpu training/inference.

A.2 Inspecting VitMTSC

Transformer self-attention layers assign an attention value between each pair of tokens. In NLP, a token is typically a word or a word-part. For images, each token represents an image patch, while in the VitMTSC model, each token represents a single MTSC datapoint in the time series. Each token is linked to the [`class`] token via the self-attention mechanism. The strength of the attention link can intuitively indicate each token's contribution to the classification. However, the attention values only represent a single facet of the Transformer network. Additionally, non-positive activation functions and skip connections impede accurate interpretation. Chefer et al. (2020) proposed attributing relevance to features in Transformer models by introducing a relevancy propagation rule that is applicable for both positive and negative transformations, normalizing relevance for non-parametric layers such as ”add” (e.g., skip-connection) and matrix multiplication, integrating the attention and relevancy scores, and combining the integrated results for multiple attention blocks.

We adapted the ViT implementation of the Chefer et al. (2020) to VitMTSC and demonstrated interpretable model outcomes. We focus on the k time-series datapoints that contributed the most to the prediction. This method identifies the top k time-series data points in a sample that contribute to a specific classification. Figure 4, for instance, depicts the top three product purchases that were significant for classifying whether a user would be receptive to a small business-oriented marketing campaign. Note that each of the three purchases consists of bulk business-related products.

A.3 UEA Datasets

The UEA MTSC archive is a repository of benchmark datasets for time-series classification (Bagnall et al., 2018). It includes 30 multivariate datasets, four of which have variable lengths. We test our model on the following five datasets (Table 1) available in the UEA repository, which include three datasets of variable length, a condition frequently observed in real-world data.

A.3.1 PenDigits

A UCI Archive dataset (Dua & Graff, 2017; Alimoglu, 1996). This is a handwritten digit classification task. 44 writers were asked to draw the digits [0...9], where instances are made up of the x and y coordinates of the pen traced across a digital screen. The data was originally recording these coordinates at a 500x500 pixel resolution, however was normalised and sampled to 100x100. Then, on expert knowledge from the dataset's authors, the data was spatially resampled such that instead of each consecutive attribute having a constant time step (100ms) but variable spatial step, they instead had constant spatial step and variable time step. From experimentation by the authors, the data was resampled to 8 spatial points, such that each instance is 2 dimenions of 8 points, with a single class label [0...9] being the digit drawn.

A.3.2 SpokenArabicDigits

This data set is taken from the UCI repository (Dua & Graff, 2017; Hammami & Sellam, 2009). It is derived from sound. Dataset from 8800 (10 digits x 10 repetitions x 88 speakers) time series of 13 Frequency Cepstral Coefficients (MFCCs) had taken from 44 males and 44 females Arabic native speakers between the ages 18 and 40 to represent ten spoken Arabic digit. Each line on the data base represents 13 MFCCs coefficients in the increasing order separated by spaces. This corresponds to one analysis frame.

A.3.3 CharacterTrajectories

From the UCI archive (Dua & Graff, 2017). The characters here were used for a PhD study on primitive extraction using HMM based models. The data consists of 2858 character samples. The data was captured using a WACOM tablet. 3 Dimensions were kept - x, y, and pen tip force. The data has been numerically differentiated and Gaussian smoothed, with a sigma value of 2. Data was captured at 200Hz. The data was normalised. Only characters with a single 'PEN-DOWN' segment were considered. Character segmentation was performed using a pen tip force cut-off point. The characters have also been shifted so that their velocity profiles best match the mean of the set. Each instance is a 3-dimensional pen tip velocity trajectory. The original data has different length cases. The class label is one of 20 characters 'a', 'b', 'c', 'd', 'e', 'g', 'h', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 'u', 'v', 'w', 'y', 'z'. To conform with the repository, we have truncated all series to the length of the shortest, which is 182, which will no doubt make classification harder.

A.3.4 FaceDetection

This data comes from a Kaggle competition in 2014 (Kaggle, 2017). The problem is to determine whether the subject has been shown the picture of a face or a scrambled image based on MEG, independent of the subject. This data set is just the training data from the competition, split by patient into 10 train subjects (subject01 to subject10) and 6 test subjects (subject11 to 16). This data cannot sensibly be randomly resampled. A leave one patient out approach is the most suitable. There are about 580-590 trials per patient, giving 5890 train trials and 3524 test subjects. Each trial consists of 1.5 seconds of MEG recording (starting 0.5sec before the stimulus starts) and the related class label, Face (class 1) or Scramble (class 0). The data were down-sampled to 250Hz and high-pass filtered at 1Hz, giving 62 observations per channel. 306 timeseries were recorded, one for each of the 306 channels, for each trial.

A.3.5 InsectWingbeat

The InsectWingbeat data was generated by the UCR computational entomology group (Chen & Adena Why, 2017) and used in the paper "Flying Insect Classification with Inexpensive Sensors. Journal of Insect Behaviour 2014" (Chen et al., 2014). The original data is a reconstruction of the sound of insects passing through a sensor. The data in the archive is the power spectrum of the sound. The 10 classes are male and female mosquitos (Ae. aegypti, Cx. tarsalis, Cx. quinquefasciants, Cx. stigmatosoma), two types of flies (Musca domestica and Drosophila simulans) and other insects. A spectorgram of each 1 second sound segment was created with a window length of 0.061 seconds and an overlap of 70%. Each instance in this multivariate dataset is arranged such that each dimension is a frequency band from the spectrogram. Each of the 10 classes in this dataset consist of 5,000 instances.

A.4 PyTorch/Pytorch Lightning code for VitMTSC

A.4.1 Imports

```
import torch
import pytorch_lightning as pl
from torch import nn
from torch.nn import functional as F
from torchmetrics import functional as FM
from petastorm import make_batch_reader
from petastorm.pytorch import DataLoader
from einops import rearrange, repeat
```

A.4.2 Residual connection

```
class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()

        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs) + x
```

A.4.3 PreNorm: Layer-Normalization

```
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()

        self.norm = nn.LayerNorm(dim)
        self.fn = fn

    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)
```

A.4.4 Feed Forward Network sub-Layer

```
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout=0.0):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)
```

A.4.5 Multi-Head Attention sub-Layer

```
class Attention(nn.Module):
    def __init__(self, dim, heads=10, dim_head=32, dropout=0.0):
        super().__init__()

        inner_dim = dim_head * heads
        self.heads = heads
        self.scale = dim_head ** -0.5

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias=False)
        self.to_out = nn.Sequential(nn.Linear(inner_dim, dim),
         ↪   nn.Dropout(dropout))

    def forward(self, x, mask=None, register_hook=False):
        b, n, _, h = *x.shape, self.heads
        qkv = self.to_qkv(x).chunk(3, dim=-1)
        q, k, v = map(lambda t: rearrange(t, "b n (h d) -> b h n
         ↪   d", h=h), qkv)

        dots = torch.einsum("bhid,bhjd->bhij", q, k) * self.scale
        mask_value = -torch.finfo(dots.dtype).max

        if mask is not None:
            mask = F.pad(mask.flatten(1), (1, 0), value=True)
            mask = mask.unsqueeze(1).unsqueeze(2)

            assert mask.shape[-1] == dots.shape[-1], "mask has
             ↪   incorrect dimensions"
            dots.masked_fill_(mask == 0.0, mask_value)
```

```
                del mask

            attn = dots.softmax(dim=-1)
            out = torch.einsum("bhij,bhjd->bhid", attn, v)

            out = rearrange(out, "b h n d -> b n (h d)")
            out = self.to_out(out)
            return out
```

A.4.6 Transformer Encoder

```
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim,
    ↪   dropout):
        super().__init__()

        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Residual(PreNorm(dim, Attention(dim, heads =
                ↪   heads, dim_head = dim_head, dropout =
                ↪   dropout))),
                Residual(PreNorm(dim, FeedForward(dim, mlp_dim,
                ↪   dropout = dropout)))
            ]))

    def forward(self, x, mask = None, register_hook = False):
        for attn, ff in self.layers:
            x = attn(x, mask = mask, register_hook =
            ↪   register_hook)
            x = ff(x)
        return x
```

A.4.7 VitMTSC model

```
class VitMTSCModel(pl.LightningModule):
    def __init__(self, config, c_in = NUMBER_OF_FEATURES, c_out =
    ↪   NUM_TARGET,
                  seq_len = SEQUENCE_LENGTH,class_weight =
                  ↪   torch.FloatTensor(CLASS_WEIGHT)):
        super(VitMTSCModel, self).__init__()

        self.d_model = config["d_model"]
        self.depth = config["depth"]
        self.heads = config["heads"]
        self.mlp_dim = config["mlp_dim"]
        self.dim_head = config["dim_head"]
        self.dropout_p = config["dropout"]
        self.emb_dropout_p = config["emb_dropout"]
        self.lr = config["lr"]
        self.weight_decay = config["weight_decay"]
        self.patience = config["patience"]

        self.pos_embedding = nn.Parameter(torch.randn(1, seq_len +
        ↪   1, self.d_model))
        self.patch_to_embedding = nn.Linear(c_in, self.d_model)
        self.cls_token = nn.Parameter(torch.randn(1, 1,
        ↪   self.d_model))
        self.dropout = nn.Dropout(self.emb_dropout_p)
        self.transformer = Transformer(self.d_model, self.depth,
         ↪  self.heads, self.dim_head, self.mlp_dim,
         ↪  self.dropout_p)
        self.to_cls_token = nn.Identity()
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(self.d_model),
            nn.Linear(self.d_model, self.mlp_dim),
            nn.GELU(),
            nn.Dropout(self.dropout_p),
            nn.Linear(self.mlp_dim, c_out)
        )

        self.c_out = c_out
        self.register_buffer('class_weight', class_weight)

    def forward(self, x, mask = None, register_hook = False):
        x = self.patch_to_embedding(x) # bs x seq_len x nvars ->
         ↪  bs x seq_len x d_model
        b, n, _ = x.shape # bs, seq_len

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b =
         ↪  b) # bs x 1 x d_model
        x = torch.cat((cls_tokens, x), dim=1) # bs x (seq_len + 1)
         ↪  x d_model
        x += self.pos_embedding[:, :(n + 1)] # +=  1 x (seq_len +
         ↪  1) x d_model -> # bs x (seq_len + 1) x d_model
        x = self.dropout(x) # bs x (seq_len + 1) x d_model

        x = self.transformer(x, mask = mask, register_hook =
         ↪  register_hook) # bs x (seq_len + 1) x d_model

        x = self.to_cls_token(x[:, 0]) # bs x d_model
        return self.mlp_head(x) # bs x num_classes

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.parameters(),
         ↪  lr=self.lr, weight_decay=self.weight_decay)
        scheduler =
         ↪  torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
         ↪  patience=self.patience)
        return {"optimizer": optimizer, "lr_scheduler": scheduler,
         ↪  "monitor": "train_loss"}

    def training_step(self, batch, batch_idx):
        x, y, _, mask = batch
        y_hat = self(x, mask)
        y = y.long()
        train_loss = F.cross_entropy(y_hat, y, weight =
         ↪  self.class_weight)
        train_auc =  FM.accuracy(F.softmax(y_hat, dim=1), y,
         ↪  num_classes = self.c_out)
        self.log('train_loss', train_loss, on_step=False,
         ↪  on_epoch=True, prog_bar=True, logger=True)
        self.log('train_auc', train_auc, on_step=False,
         ↪  on_epoch=True, prog_bar=True, logger=True)
        return train_loss

    def validation_step(self, batch, batch_idx):
        x, y, _, mask = batch
        y_hat = self(x, mask)
        y = y.long()
        val_loss = F.cross_entropy(y_hat, y, weight =
        ↪  self.class_weight)
        val_auc =  FM.accuracy(F.softmax(y_hat, dim=1), y,
        ↪  num_classes = self.c_out)
        self.log('val_loss', val_loss, on_step=False,
        ↪  on_epoch=True, prog_bar=True, logger=True,
        ↪  sync_dist=True)
        self.log('val_auc', val_auc, on_step=False, on_epoch=True,
        ↪  prog_bar=True, logger=True)
        return val_loss

    def test_step(self, batch, batch_idx):
        x, y, _, mask = batch
        y_hat = self(x, mask)
        y = y.long()
        test_loss = F.cross_entropy(y_hat, y, weight =
        ↪  self.class_weight)
        test_auc =  FM.accuracy(F.softmax(y_hat, dim=1), y,
        ↪  num_classes = self.c_out)
        self.log('test_loss', test_loss, on_step=False,
        ↪  on_epoch=True, prog_bar=True, logger=True,
        ↪  sync_dist=True)
        self.log('test_auc', test_auc, on_step=False,
        ↪  on_epoch=True, prog_bar=True, logger=True)
        return test_loss
```

A.4.8 VITMTSC: PETASTORM DATAMODULE

```
def petastorm_collate_fn(rows):
    data_df = pd.DataFrame(rows)
    case_id_df = data_df.iloc[:,
    ↪  NUMBER_OF_FEATURES*SEQUENCE_LENGTH+1
        :NUMBER_OF_FEATURES*SEQUENCE_LENGTH+2]
    case_id_tensor =
    ↪  torch.tensor(case_id_df.values.astype(np.float64))

    target_df = data_df.iloc[:,
    ↪  NUMBER_OF_FEATURES*SEQUENCE_LENGTH+0
        :NUMBER_OF_FEATURES*SEQUENCE_LENGTH+1]
    target_tensor =
    ↪  torch.tensor(target_df.values.astype(np.float32))

    data_tensor_df = data_df.iloc[:, 0*SEQUENCE_LENGTH
        :NUMBER_OF_FEATURES*SEQUENCE_LENGTH]
    data_tensor =
    ↪  torch.tensor(data_tensor_df.values.astype(np.float32))
    data_tensor = rearrange(data_tensor, 't (b h)-> t h b', h =
    ↪  SEQUENCE_LENGTH)

    mask_df = data_df.iloc[:, 0*SEQUENCE_LENGTH
        :1*SEQUENCE_LENGTH]
    mask_tensor = torch.tensor(mask_df.values.astype(np.float32))

    return data_tensor, target_tensor.squeeze(),
    ↪  case_id_tensor.squeeze(), mask_tensor.squeeze()

class VitMTSCPetastormDataModule(pl.LightningDataModule):
    def __init__(
        self,
        config,
        train_files,
        valid_files,
        test_files,
        num_workers=NUM_WORKERS,
        transform_spec=None,
        shard_count=NUM_GPUS,
        num_epochs=MAX_EPOCHS,
    ):
        super().__init__()
        self.train_files = train_files
        self.valid_files = valid_files
        self.test_files = test_files
        self.batch_size = config["batch_size"]
        self.num_workers = num_workers
        self.transform_spec = transform_spec
        self.shard_count = shard_count
        self.num_epochs = num_epochs

    def train_dataloader(self):

        self.train_ds = make_batch_reader(
            self.train_files,
            workers_count=self.num_workers,
            transform_spec=self.transform_spec,
            cur_shard=int(os.environ["LOCAL_RANK"]),
            shard_count=self.shard_count,
            num_epochs=self.num_epochs,
        )
        return DataLoader(self.train_ds,
         ↪  batch_size=self.batch_size,
         ↪  collate_fn=petastorm_collate_fn)

    def val_dataloader(self):
        print(
            "val_dataloader: local rank :",
            int(os.environ["LOCAL_RANK"]),
            "shard count: ",
            self.shard_count,
        )
        self.val_ds = make_batch_reader(
            self.valid_files,
            workers_count=self.num_workers,
            transform_spec=self.transform_spec,
            cur_shard=int(os.environ["LOCAL_RANK"]),
            shard_count=self.shard_count,
            num_epochs=self.num_epochs,
        )
        return DataLoader(self.val_ds, batch_size=self.batch_size,
         ↪  collate_fn=petastorm_collate_fn)

    def test_dataloader(self):
        print(
            "test_dataloader: local rank :",
            int(os.environ["LOCAL_RANK"]),
            "shard count: ",
            self.shard_count,
        )
```

```
self.test_ds = make_batch_reader(
    self.test_files,
    workers_count=self.num_workers,
    transform_spec=self.transform_spec,
    cur_shard=int(os.environ["LOCAL_RANK"]),
    shard_count=self.shard_count,
    num_epochs=self.num_epochs,
)
return DataLoader(self.test_ds,
 ↪  batch_size=self.batch_size,
 ↪  collate_fn=petastorm_collate_fn)
```